Pascal Pons

Follow

SOLVING CONNECT FOUR

Part 9 – Anticipate direct losing moves

The idea is to anticipate and avoid exploring very bad moves allowing the opponent to win directly at the next turn. That way we are able to prune the search tree faster and reduce the number of explored nodes.

To implement efficiently this move anticipation we have to identify opponent's winning positions. Except if we have a direct winning move, the rules to avoid playing losing moves are:

- We should always play a column on which the opponent has a winning position in the bottom of the column.
- We should never play under an opponent winning positions.
- If the opponent has more than two directly playable winning positions, then we cannot do anything and we will lose.

Implementation

The implementation mainly rely on a new function possibleNonLosingMoves() providing a bitmap of all possible next playable positions that do not make the opponent win directly at the next move. The function implements the 3 rules identifying the non losing possible moves.

```
/*
 * Return a bitmap of all the possible next moves the do not lose in one turn.
 * A losing move is a move leaving the possibility for the opponent to win directly.
 *
 * Warning this function is intended to test position where you cannot win in one turn
 * If you have a winning move, this function can miss it and prefer to prevent the opponent
 * to make an alignment.
```

```
   */
  uint64_t possibleNonLosingMoves() const {
    assert(!canWinNext());
    uint64_t possible_mask = possible();
    uint64_t opponent_win = opponent_winning_position();
    uint64_t forced_moves = possible_mask & opponent_win;
      if(forced_moves) {
        if(forced_moves & (forced_moves - 1)) // check if there is more than one forced move
          return 0;                           // the opponnent has two winning moves and you cannot stop him
        else possible_mask = forced_moves;    // enforce to play the single forced move
      }
    return possible_mask & ~(opponent_win >> 1);  // avoid to play below an opponent winning spot
  }
```

The possible() function provide a bimap of all possible moves. opponent_winning_position() just call the main compute_winning_position() function that is making heavy use of bitboard bitwise operations to identify all winning positions of a given board. Meaning all open ended 3-aligments.

```
/*
 * Return a bitmask of the possible winning positions for the opponent
 */
uint64_t opponent_winning_position() const {
  return compute_winning_position(current_position ^ mask, mask);
}

uint64_t possible() const {
  return (mask + bottom_mask) & board_mask;
}

static uint64_t compute_winning_position(uint64_t position, uint64_t mask) {
  // vertical;
  uint64_t r = (position << 1) & (position << 2) & (position << 3);

  //horizontal
  uint64_t p = (position << (HEIGHT+1)) & (position << 2*(HEIGHT+1));
  r |= p & (position << 3*(HEIGHT+1));
  r |= p & (position >> (HEIGHT+1));
  p >>= 3*(HEIGHT+1);
  r |= p & (position << (HEIGHT+1));
  r |= p & (position >> 3*(HEIGHT+1));

  //diagonal 1
  p = (position << HEIGHT) & (position << 2*HEIGHT);
  r |= p & (position << 3*HEIGHT);
  r |= p & (position >> HEIGHT);
  p >>= 3*HEIGHT;
  r |= p & (position << HEIGHT);
  r |= p & (position >> 3*HEIGHT);

  //diagonal 2
  p = (position << (HEIGHT+2)) & (position << 2*(HEIGHT+2));
  r |= p & (position << 3*(HEIGHT+2));
  r |= p & (position >> (HEIGHT+2));
  p >>= 3*(HEIGHT+2);
  r |= p & (position << (HEIGHT+2));
  r |= p & (position >> 3*(HEIGHT+2));

  return r & (board_mask ^ mask);
}
```

The implementation in the negamax function is quite straitforward. Note that now we will never explore a move that makes the opponent win directly, thus we no longer have to check if the current player can win directly, saving some time.

Full source code corresponding to this part.

Benchmark

Anticipating one move in advance reduces the number of explored nodes by allowing to prune the search earlier. Meanwhile, identifiying the non-losing moves is an extra additional computation increasing the average computation time per node. Fortunatlely the bitboard implementation is quite efficient and allows to compute all possible non-losing moves quite fast.

Solver	Test Set name	mean time	mean nb pos	K pos/s
Skipping losing moves (strong solver)	End-Easy	4.606 μs	70.71	15,350
Skipping losing moves (strong solver)	Middle-Easy	124.4 μs	4,135	33,230
Skipping losing moves (strong solver)	Middle-Medium	32.37 ms	1,135,000	35,070
Skipping losing moves (strong solver)	Begin-Easy	3.505 ms	107,400	30,630
Skipping losing moves (strong solver)	Begin-Medium	2.758 s	110,800,000	40,150
Skipping losing moves (strong solver)	Begin-Hard	N/A	N/A	N/A
Skipping losing moves (weak solver)	End-Easy	3.568 μs	43.30	12,140
Skipping losing moves (weak solver)	Middle-Easy	736.8 μs	19,800	26,870
Skipping losing moves (weak solver)	Middle-Medium	17.55 ms	564,600	32,170
Skipping losing moves (weak solver)	Begin-Easy	829.5 ms	27,010,000	32,560
Skipping losing moves (weak solver)	Begin-Medium	1.265 s	44,860,000	35,460
Skipping losing moves (weak solver)	Begin-Hard	N/A	N/A	N/A

Tutorial plan

SOLVING CONNECT FOUR

Updated: September 09, 2017

Previous | Next